PROJECT 3
STRUCTURED LIGHT AND POINT CLOUD DATA

OVERVIEW

In this lab you will investigate how the first-generation Microsoft Kinect sensor uses structured light to generate dense range data, and learn how to reconstruct 3D point clouds from such data.

TASKS

Background. Before you begin, read the ROS Kinect technical documentation at http://wiki.ros.org/kinect_calibration/technical to get an idea of how the first-generation Kinect works.

Getting started. Begin by downloading the project 3 starter code and files from the course website. Inside the archive, you will find a projector image and a number of simulated camera images of 3D objects built using a simplified model of a Kinect sensor. There are also two Python scripts, **starter.py** and **PointCloudApp.py** which help compute disparity maps and visualize 3D data.[1]

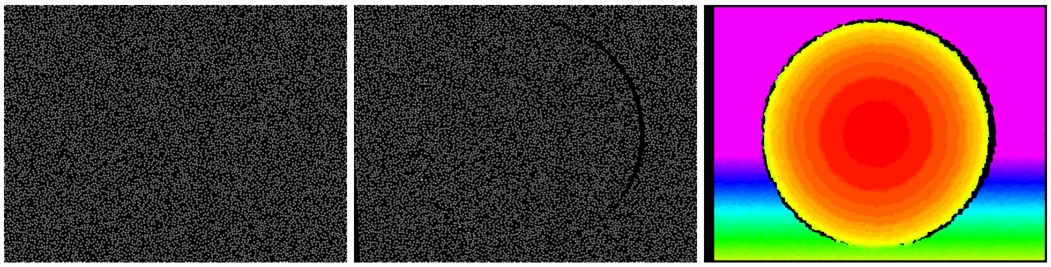
You can test the **PointCloudApp** module by running one of the two commands

```
python PointCloudApp.py
python PointCloudApp.py cam1_xyz.npz
```

The first should display a 3D torus, and the second should display the 3D reconstruction corresponding to the **cam1.png** file. In either case, you can click and drag in the 3D window to rotate the view.

Dense point cloud reconstruction. Given the projector image, and any camera image, you can compute a *disparity image* for the scene. OpenCV makes this very easy via the **StereoSGBM** object (see http://goo.gl/U5iW51).

[1]You may also want to install the OpenGL module for Python to get nicer performance from PointCloudApp. This is straightforward on Mac or Linux, but I have no clue how it's done on Windows. At any rate, it works fine without OpenGL.


Left to right: projector image, simulated IR camera image, and disparity map.

Your goal is to extend the **`starter.py`** program to generate an n-by-3 array of XYZ data to be used as input to the **`PointCloudApp`** module, where n is the useable number of pixels from the disparity image.

To do so, you will need to consider the camera geometry as well as the intrinsic parameters. For the simulated setup, the calibration matrices $\boldsymbol{K}$ for both the projector and the camera are identical, with

$$K = \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 600 & 0 & 320 \\ 0 & 600 & 240 \\ 0 & 0 & 1 \end{bmatrix}$$

and b, the stereo baseline, is 0.05 m.

Remember, $\boldsymbol{K}$ maps a 3D point $\boldsymbol{P}$ in the frame of the camera to a point on the sensor plane:

$$\boldsymbol{q} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \boldsymbol{K} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Hence, mapping each point $\boldsymbol{q}$ through $\boldsymbol{K}^{-1}$ will return some point $\boldsymbol{P}$ which is proportional to (X, Y, Z). Furthermore, the Z coordinate of each point $\boldsymbol{P}$ can be obtained by examining the disparity value Δ at each (u, v) location via

$$Z = \frac{b \cdot f}{\Delta}$$

Using this knowledge, it is possible to reconstruct the (X, Y, Z) location for every pixel in the disparity image where $\Delta \neq 0$.[2]

[2]In practice, it is best to restrict this to $\Delta > bf/Z_{max}$ for some suitably large value of Z_{max}. For this project, I suggest using Z_{max} = 8 m.

Vectorized code. Due to the inefficiency of interpreted languages, and the ability of underlying libraries to exploit data parallelism, programs in environments like MATLAB and **`numpy`** often perform much better if they operate on entire arrays rather than using **`for`** loops to explicitly iterate over the data.[3] Although such *vectorized* implementations are faster, they can sometimes be more difficult to write.

Your final program should be vectorized, containing no explicit iteration (i.e. loops). The bottom part of **`PointCloudApp.py`** contains some helpful examples of array operations that avoid loops – in particular, pay attention to the use of **`numpy.meshgrid`** and the use of logical masks for array indexing.

You might find it helpful to write and debug an iterative version of your program before writing the vectorized version. For this problem, note that $\boldsymbol{K}^{-1}$ has a very simple form, which might help guide your implmentation.

WHAT TO TURN IN

In addition to your program source code, you should submit a PDF writeup that addresses the following topics:

1. How did you approach writing the vectorized version of the program? Did you implement the iterative version first and then modify it, or did you do something else?

2. Vary the **`window_size`** parameter for stereo matching. Try values of 7, 9, 15, and 21. How do the point clouds change? Are there any benefits to using smaller values? Larger ones?

3. In addition to the projector and IR camera, Kinect sensors also have an RGB camera which records color images. This can be useful for assigning colors to the points in the XYZ point cloud data (the resulting representation is sometimes referred to as XYZRGB).

   The starter code for this project includes both data taken from a real Kinect as well as a program `load_kinect_data.py` to view it. This particular data was taken from a sensor mounted on a TurtleBot 2 (http://www.turtlebot.com/). Given that knowledge, why is there a blank vertical stripe on the right hand side of each point cloud from the real Kinect? What physical feature does the stripe correspond to? (Study the robot photo on the website for hints...)

Please submit your project to the course Moodle by 11:55PM on Sunday, 3/26.

[3] See http://en.wikipedia.org/wiki/Data_parallelism.